

# ソフトウェア特論

創作情報工学研究室

平井 喜一

# お品書き

---

- 幅優先探索 p.151~159
- バックトラック法 p.159~164
- 文字列探索 p.170~177

# お品書き

---

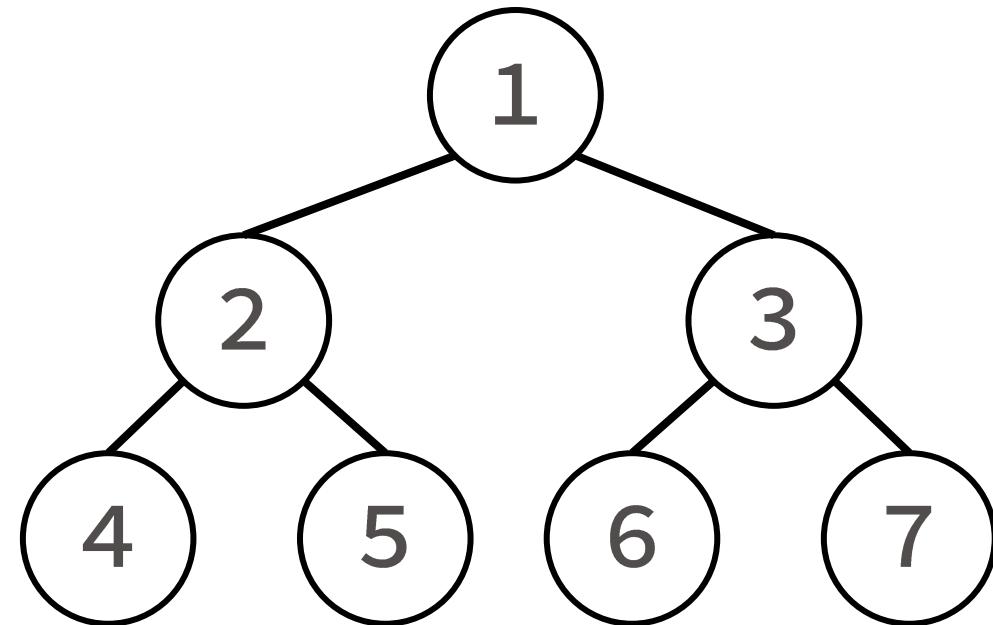
- 幅優先探索 p.151~159
- バックトラック法 p.159~164
- 文字列探索 p.170~177

# 幅優先探索(BFS)

グラフや木構造を探索するアルゴリズム

P151

探索を開始する頂点からの距離(深さ)が等しくなるように進んでいく方式



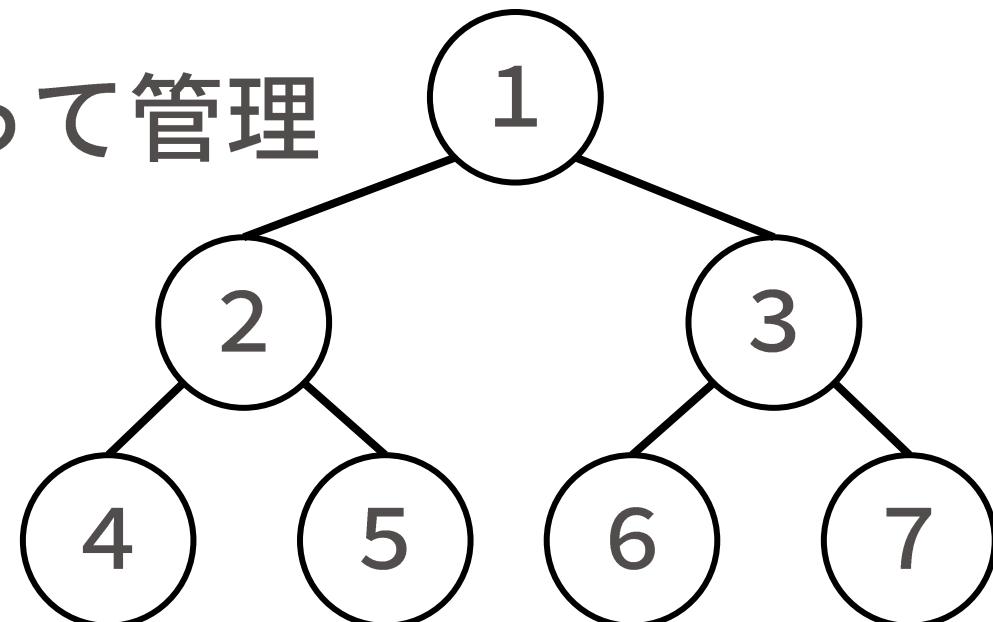
# 幅優先探索(BFS)

グラフや木構造を探索するアルゴリズム

P151

探索を開始する頂点からの距離(深さ)が等しくなるように進んでいく方式

探索するノードはキューを使って管理  
(始点から近い順に探索をしていくため)



# おさらい…キューってなんだっけ？

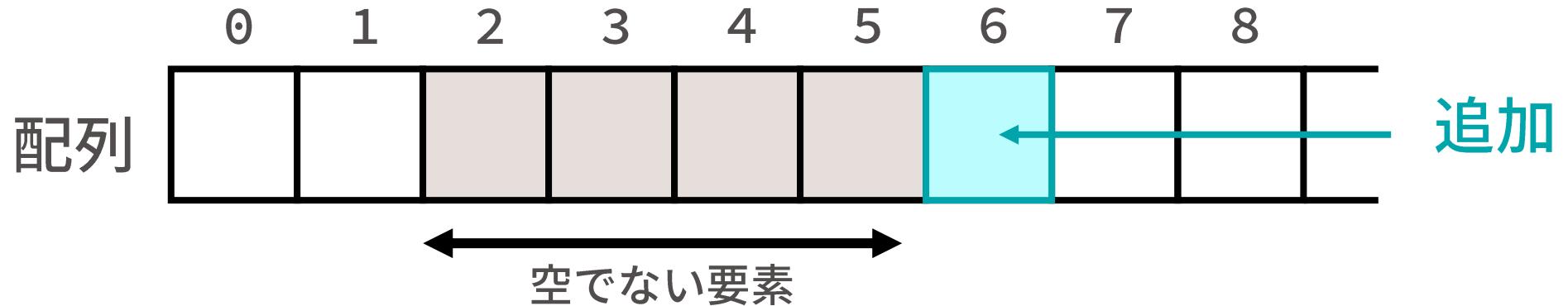
---

キューってなんだっけ？

# おさらい…キューってなんだっけ？

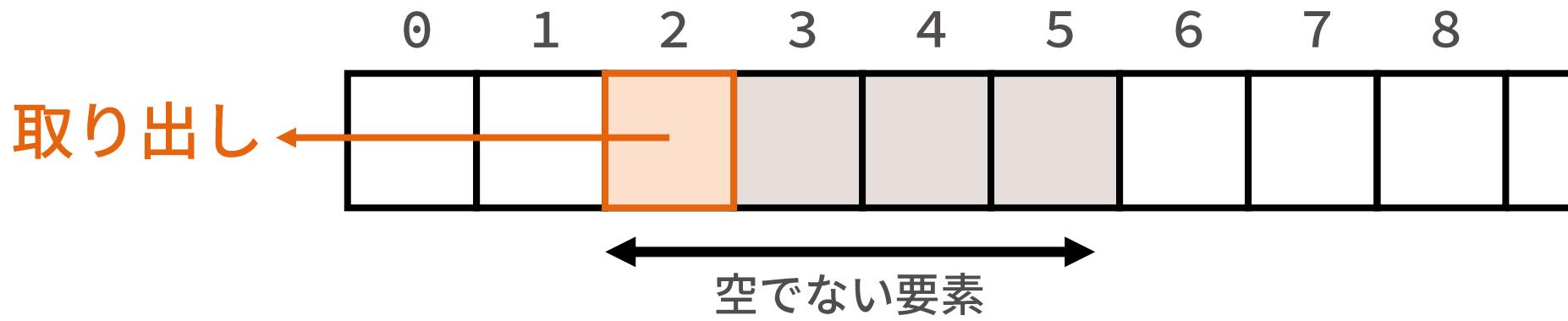
- データの追加

配列の空の要素を除いた**最後尾の位置**の1つ後ろの位置にデータを格納



- データの取り出し

配列の空の要素を除いた**先頭の位置**からデータを取得



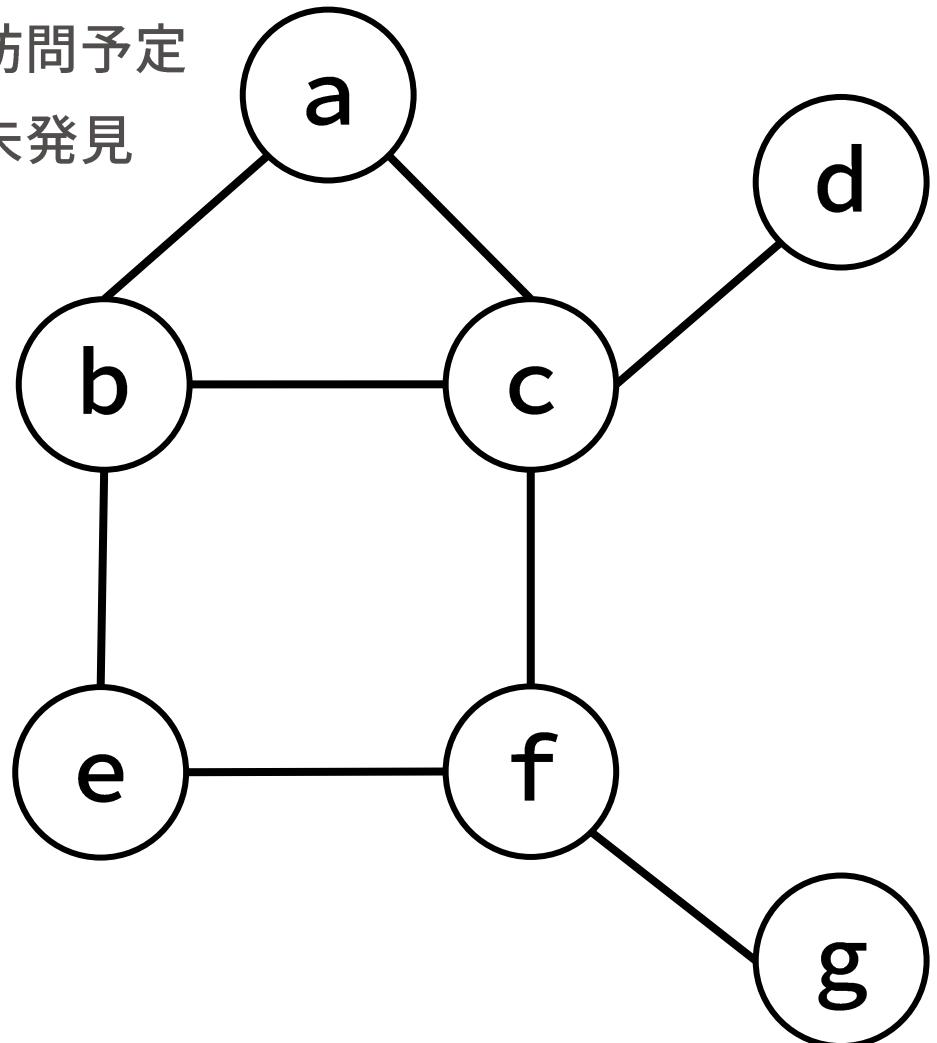
# 幅優先探索(BFS)

P153-5

赤：訪問済み

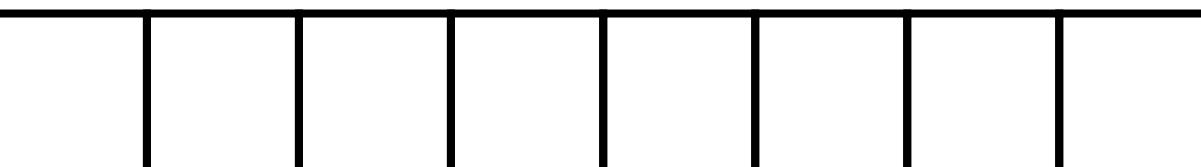
橙：訪問予定

白：未発見



	a	b	c	d	e	f	g
a	0	1	1	0	0	0	0
b	1	0	1	0	1	0	0
c	1	1	0	1	1	1	0
d	0	0	1	0	0	0	0
e	0	1	1	0	0	1	0
f	0	0	1	0	1	0	1
g	0	0	0	0	0	1	0

キュー



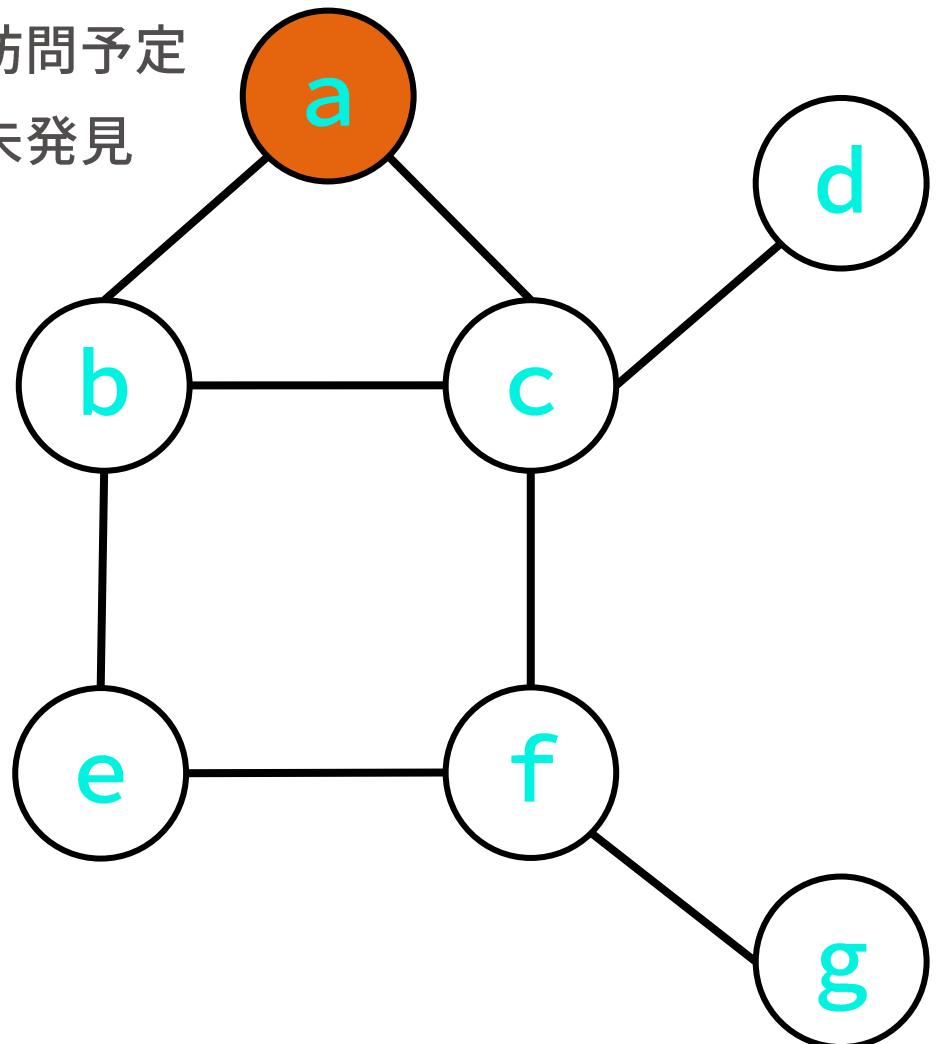
# 幅優先探索(BFS)

P153-5

赤：訪問済み

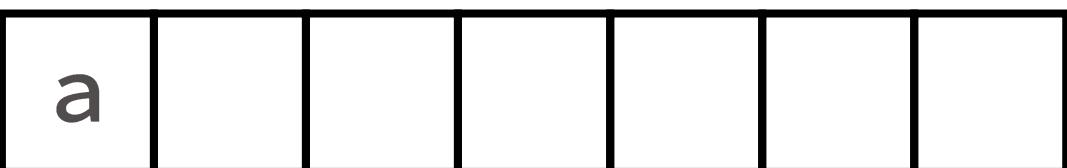
橙：訪問予定

白：未発見



	a	b	c	d	e	f	g
a	0	1	1	0	0	0	0
b	1	0	1	0	1	0	0
c	1	1	0	1	1	1	0
d	0	0	1	0	0	0	0
e	0	1	1	0	0	1	0
f	0	0	1	0	1	0	1
g	0	0	0	0	0	1	0

キュー



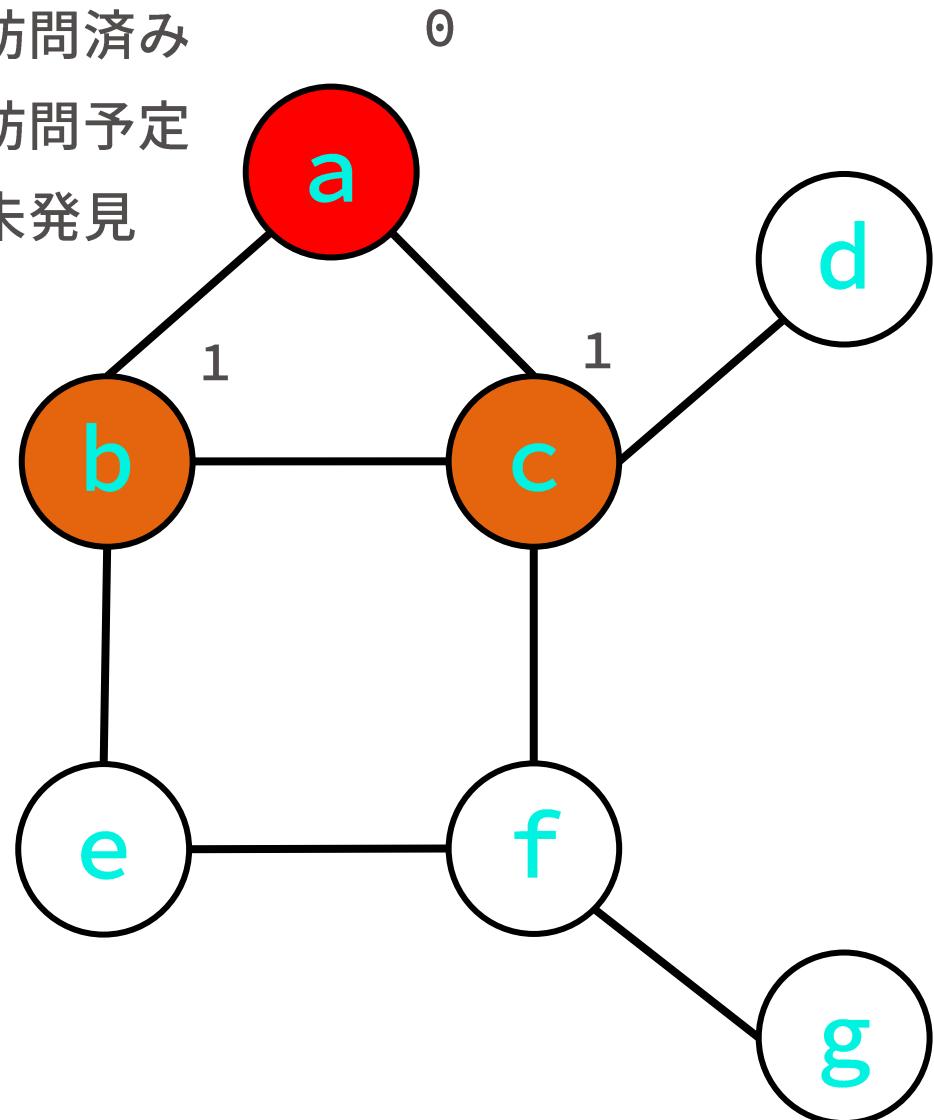
# 幅優先探索(BFS)

P153-5

赤：訪問済み

橙：訪問予定

白：未発見



探索テーブル

a	b	c	d	e	f	g
0	0	0	0	0	0	0



a	b	c	d	e	f	g
1	0	0	0	0	0	0

キュー

b	c					
---	---	--	--	--	--	--

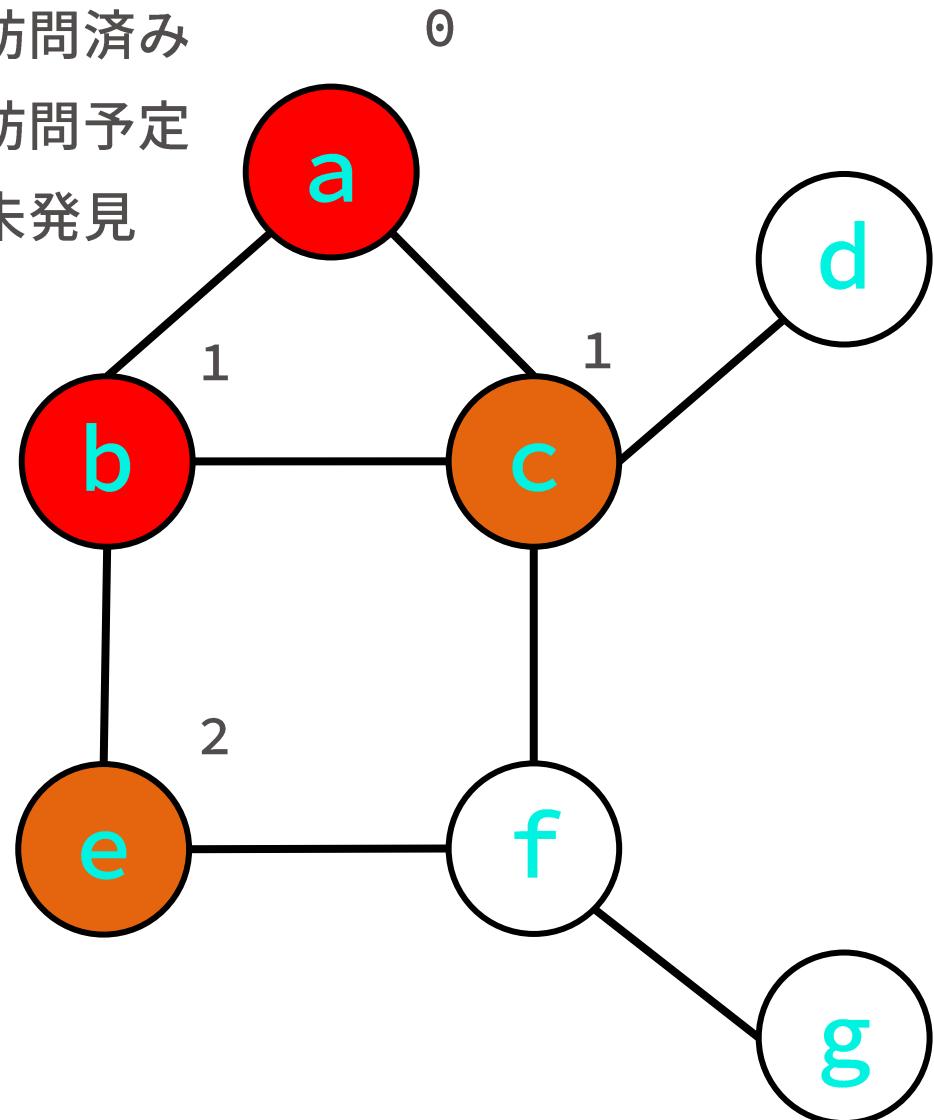
# 幅優先探索(BFS)

P153-5

赤：訪問済み

橙：訪問予定

白：未発見



探索テーブル

a	b	c	d	e	f	g
1	0	0	0	0	0	0



a	b	c	d	e	f	g
1	1	0	0	0	0	0

キュー

c	e					
---	---	--	--	--	--	--

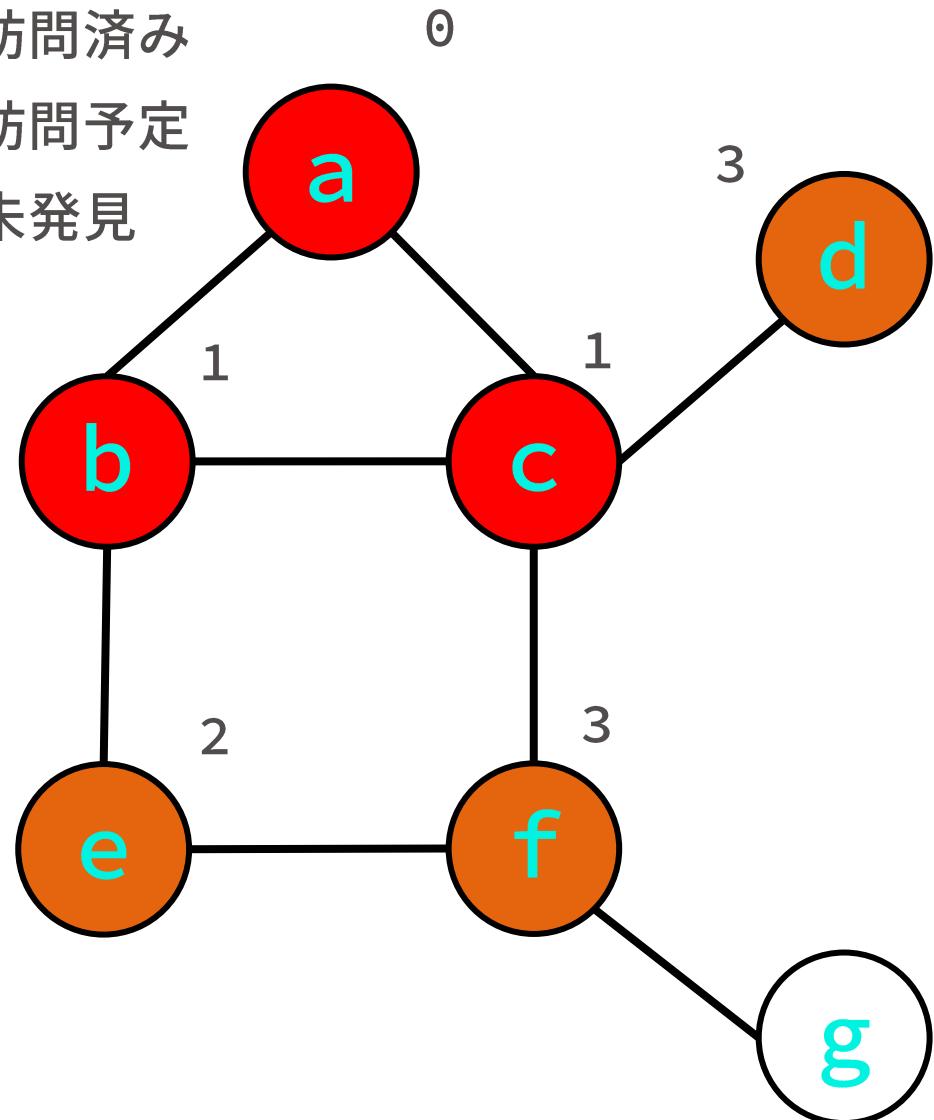
# 幅優先探索(BFS)

P153-5

赤：訪問済み

橙：訪問予定

白：未発見



探索テーブル

a	b	c	d	e	f	g
1	1	0	0	0	0	0



a b c d e f g

a	b	c	d	e	f	g
1	1	1	0	0	0	0

キュー

e	d	f				
---	---	---	--	--	--	--

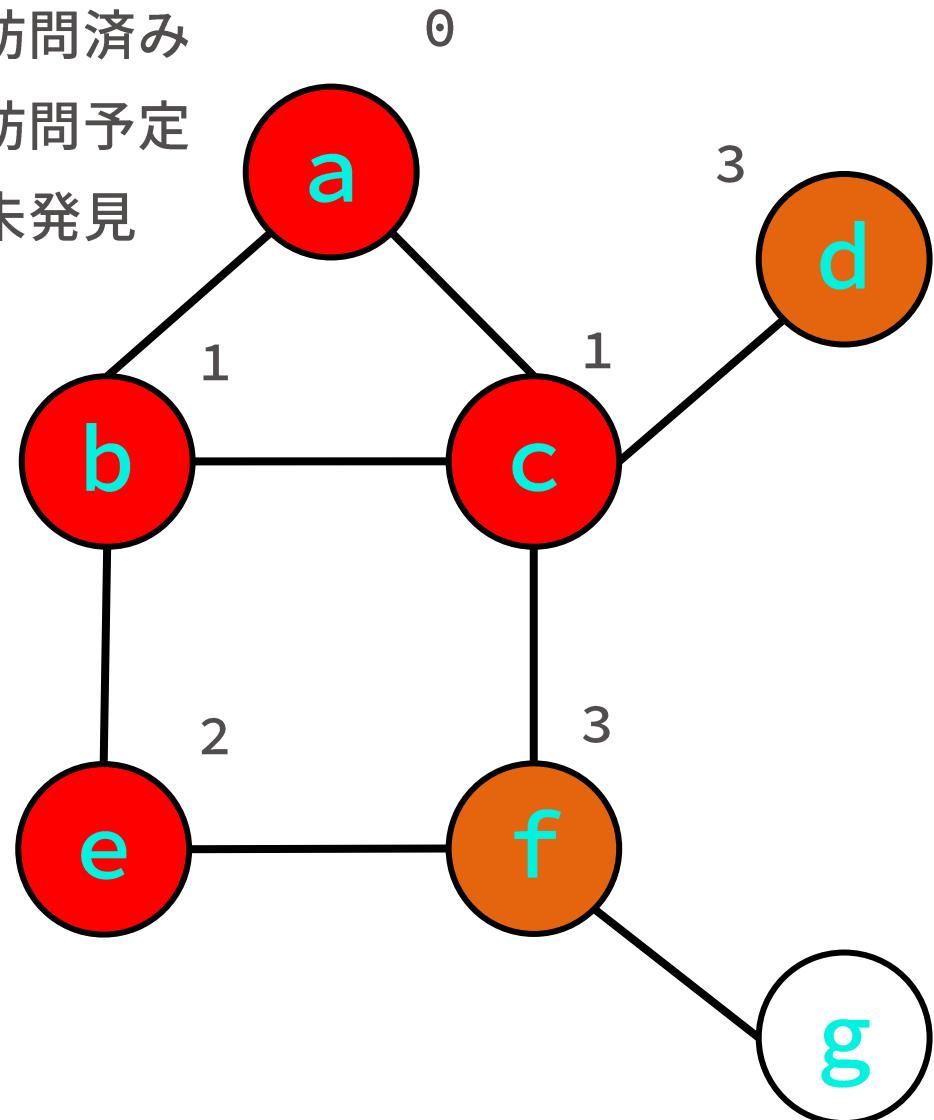
# 幅優先探索(BFS)

P153-5

赤：訪問済み

橙：訪問予定

白：未発見



探索テーブル

a	b	c	d	e	f	g
1	1	1	0	0	0	0



a	b	c	d	e	f	g
1	1	1	0	1	0	0

キュー

d	f					
---	---	--	--	--	--	--

fは発見済みなので追加なし

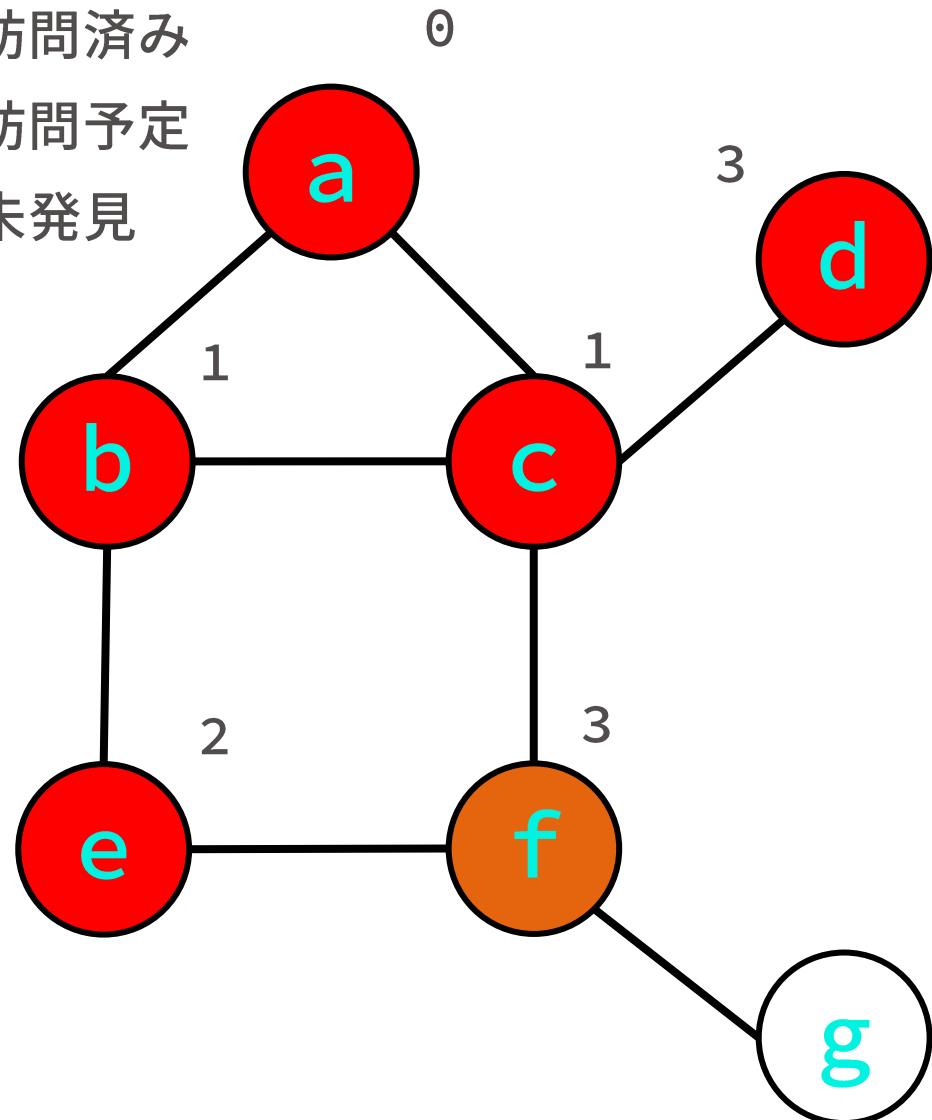
# 幅優先探索(BFS)

P153-5

赤：訪問済み

橙：訪問予定

白：未発見



探索テーブル

a	b	c	d	e	f	g
1	1	1	0	1	0	0



a	b	c	d	e	f	g
1	1	1	1	1	0	0

キュー

cは訪問済みなので追加なし

f						
---	--	--	--	--	--	--

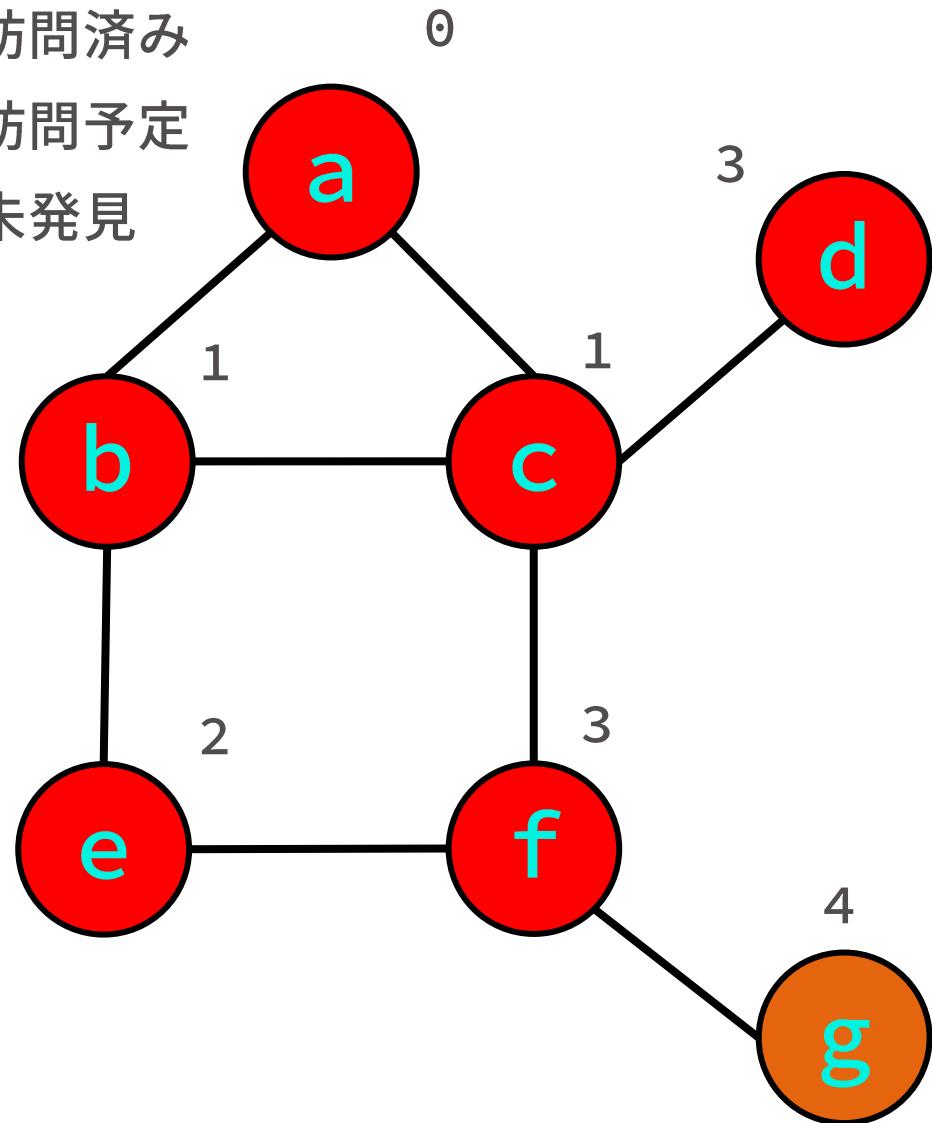
# 幅優先探索(BFS)

P153-5

赤：訪問済み

橙：訪問予定

白：未発見



探索テーブル

a	b	c	d	e	f	g
1	1	1	1	1	0	0



a	b	c	d	e	f	g
1	1	1	1	1	1	0

キュー

g						
---	--	--	--	--	--	--

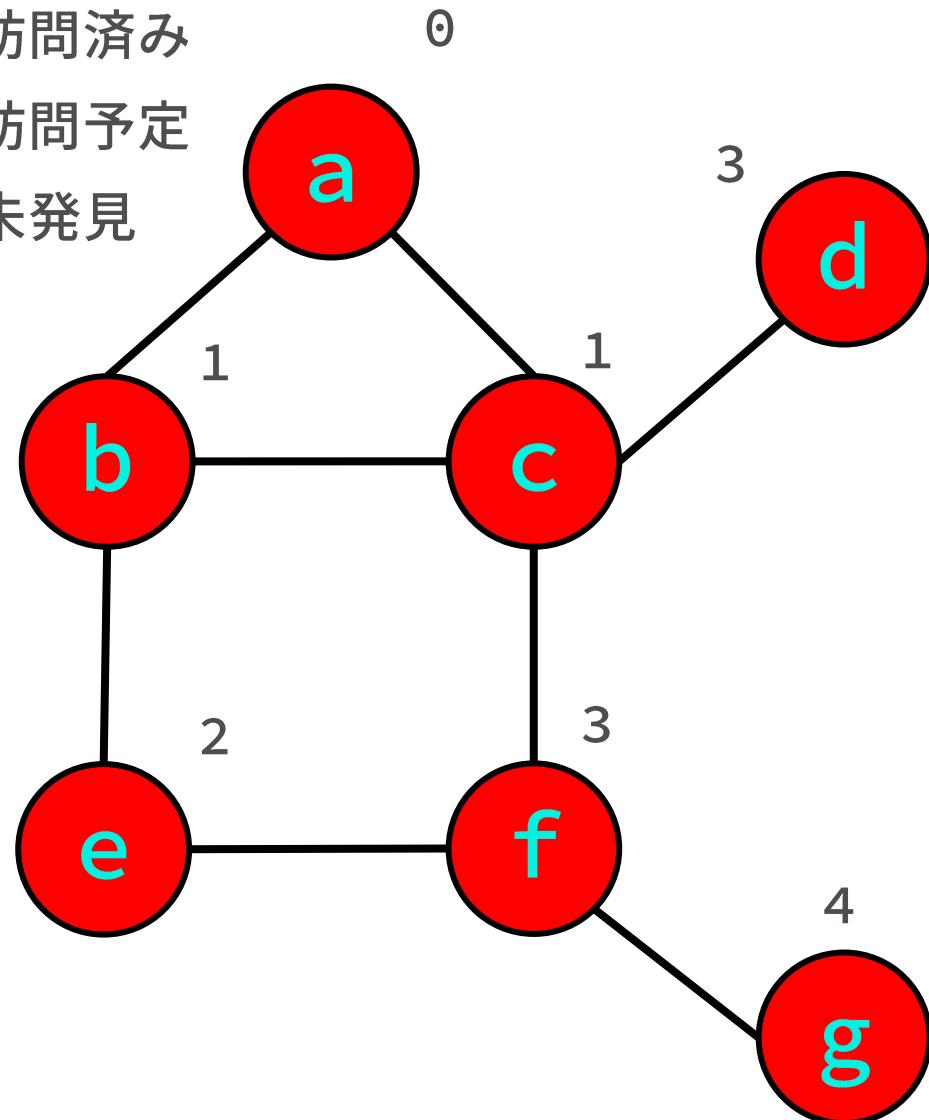
# 幅優先探索(BFS)

P153-5

赤：訪問済み

橙：訪問予定

白：未発見



探索テーブル

a	b	c	d	e	f	g
1	1	1	1	1	1	0



a b c d e f g

a	b	c	d	e	f	g
1	1	1	1	1	1	1

キュー

--	--	--	--	--	--	--

## アルゴリズム的なお話

- ① 行列の設定
- ② 探索テーブルを全て0にし，全てのノードを未探索状態に
- ③ 未探索のノードがなくなるまで④~⑤を繰り返す
- ④ 未探索のノードを1つ取り出し，キューにぶち込む
- ⑤ キューに未探索のノードが存在する間，以下の処理をする
  - ⑤-1 キューから未探索ノードを取り出す
  - ⑤-2 取り出した未探索ノードを探索済みに設定
  - ⑤-3 このノードから辺1つ隔てて，到達可能な未探索ノードを全てキューに登録
- ⑥ おわり

# 幅優先探索(BFS)

---

## 計算量

探索するグラフの頂点数をN，辺数をM

各頂点はキューに1回挿入され，1回取り出されるの

→ 計算量  $o(N)$

各辺は1回だけ探索されることになる

→ 計算量  $o(M)$

合わせて計算量は  $o(N + M)$

# 幅優先探索(BFS)

---

## 深さ優先探索との使い分け

### 深さ優先探索を使うケース

- 全通りを列挙し、結果をまとめる必要がある場合
- 文字列などを探索する時に、「辞書順最小」であることが求められている場合

### 幅優先探索を使うケース

- 始点から最も近いものを探したい場合
- 探索範囲自体は広いものの、ある程度近くに探したい解が存在することがわかっている場合
- 探索範囲が広く、深さ優先ではスタックが大量使用されてしまう場合

# お品書き

---

- 幅優先探索 p.151~159
- バックトラック法 p.159~164
- 文字列探索 p.170~177

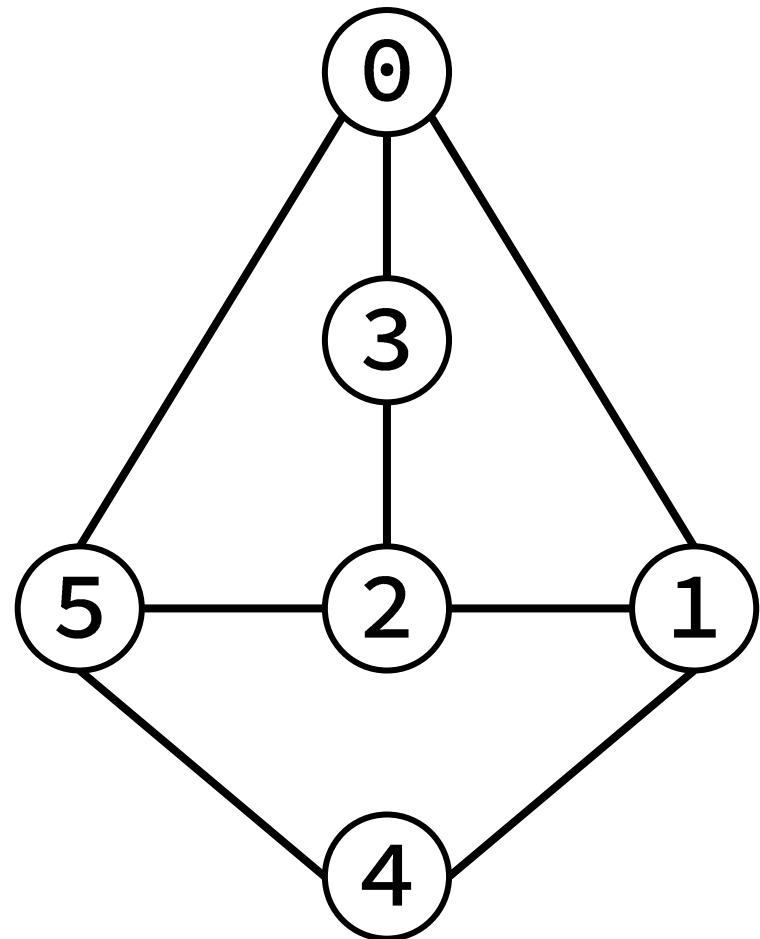
問題の解を見つけるために，**全てのパターンを系統的に探索して虱潰しに試していくが，途中で解になり得ないと分かったパターンは間引く手法**

探索の際に候補の数ができるだけ少なくおさえて，  
**深さ優先探索**により効率よく探索できる

# バックトラック法 ハミルトンの閉路問題

ハミルトンの閉路問題を用いた考え方

P159



開始ノード

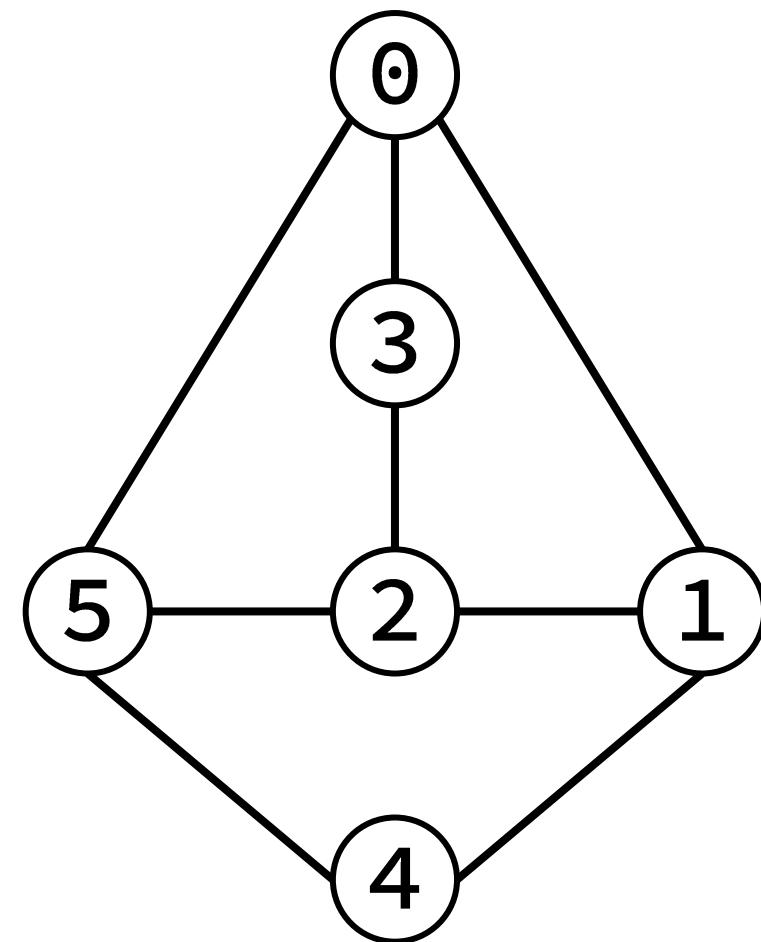
行先ノード
0 1 2 3 4 5
0 {0,1,0,1,0,1},
1 {1,0,1,0,1,0},
2 {0,1,0,1,0,1},
3 {1,0,1,0,0,1},
4 {0,1,0,0,0,1},
5 {1,0,1,1,1,0}}

# バックトラック法 ハミルトンの閉路問題

2番を出発し、全ての頂点を1度だけ訪問

P159

枝切りについて



- ① 2-3-0-5-4-1-2
- ② 2-3-0-1-4-5-2
- ③ 2-5-4-1-0-3-2
- ④ 2-1-4-5-0-3-2

# バックトラック法 ハミルトンの閉路問題

2番を出発し、全ての頂点を1度だけ訪問

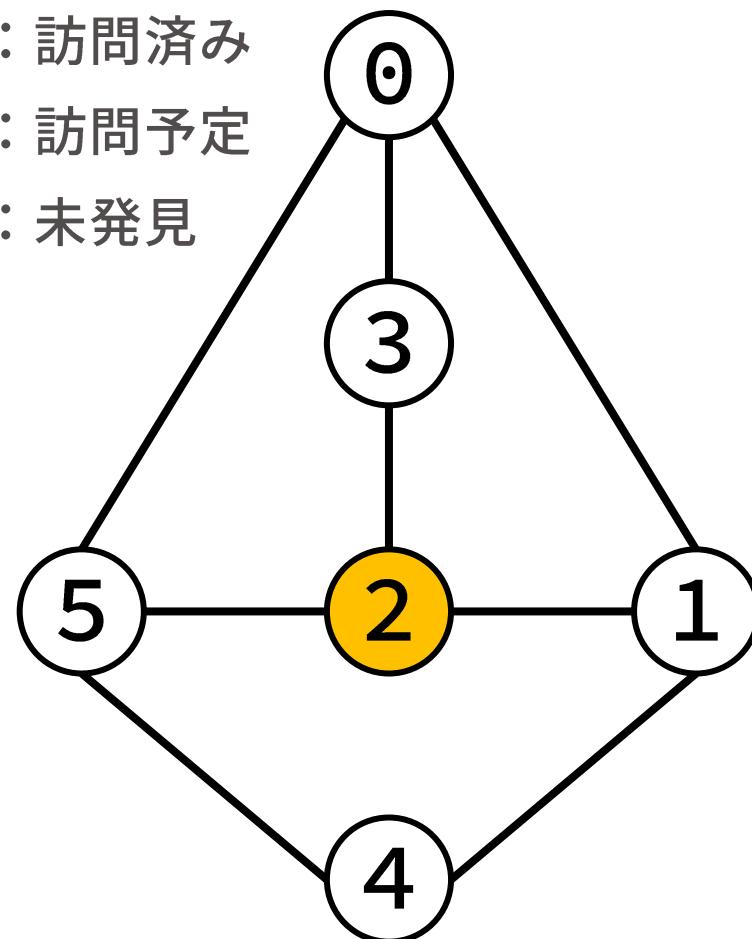
P159

枝切りについて

- (1) 2-3-0-5-4-1-2
- (2) 2-3-0-1-4-5-2

※(2)は探索済みとする

赤：訪問済み  
橙：訪問予定  
白：未発見



# バックトラック法 ハミルトンの閉路問題

2番を出発し、全ての頂点を1度だけ訪問

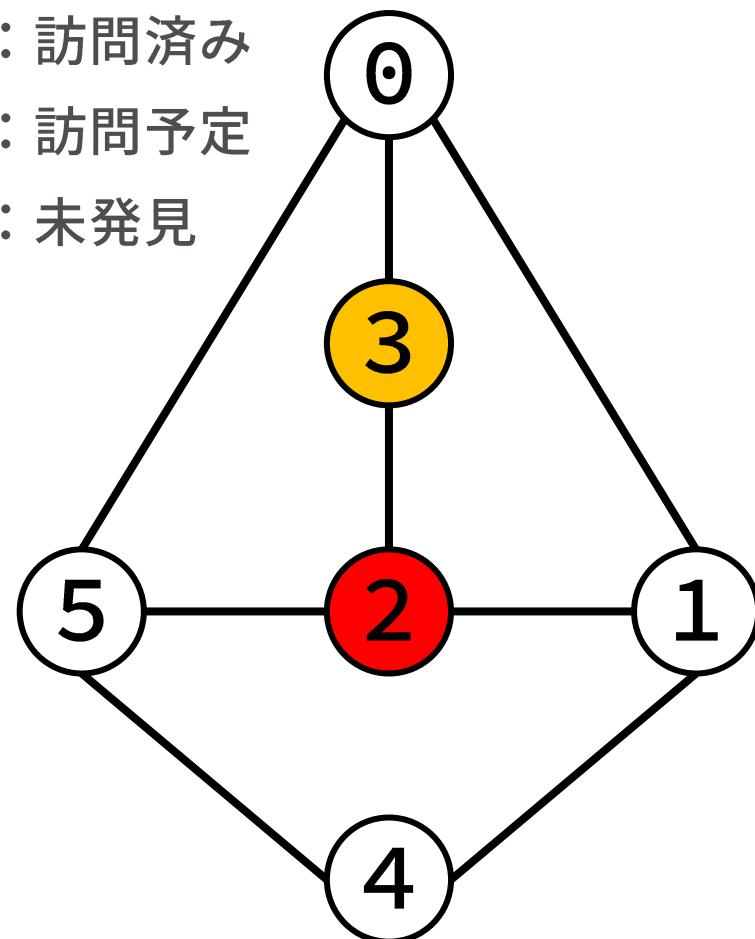
P159

枝切りについて

- (1) 2-3-0-5-4-1-2
- (2) 2-3-0-1-4-5-2

※(2)は探索済みとする

赤：訪問済み  
橙：訪問予定  
白：未発見



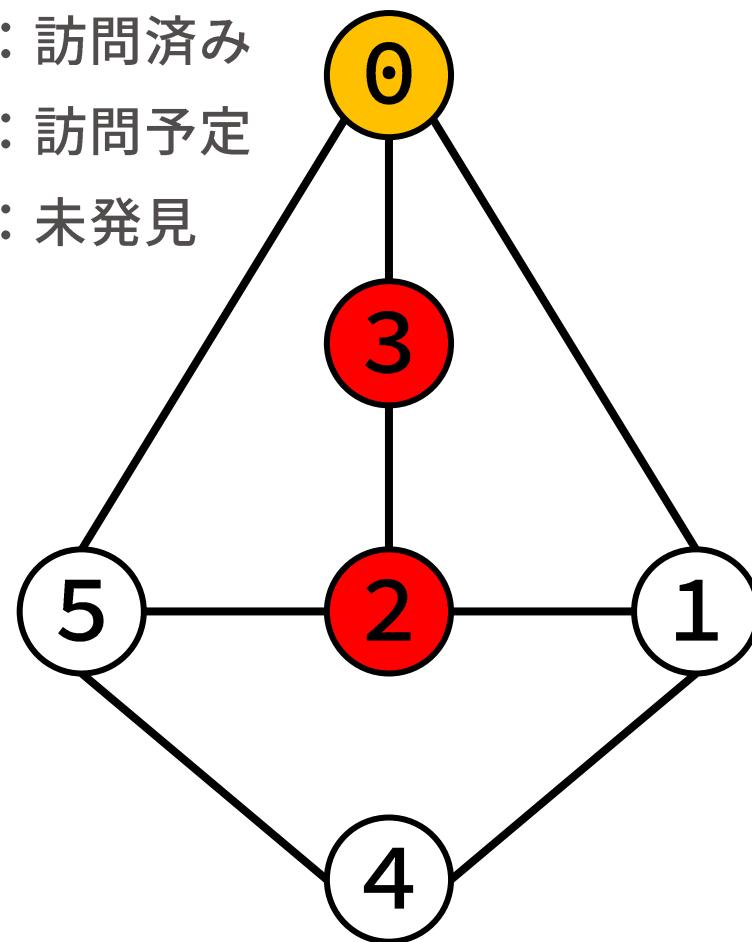
# バックトラック法 ハミルトンの閉路問題

2番を出発し、全ての頂点を1度だけ訪問

P159

枝切りについて

赤：訪問済み  
橙：訪問予定  
白：未発見



- (1) 2-3-0-5-4-1-2  
(2) 2-3-0-1-4-5-2

※(2)は探索済みとする

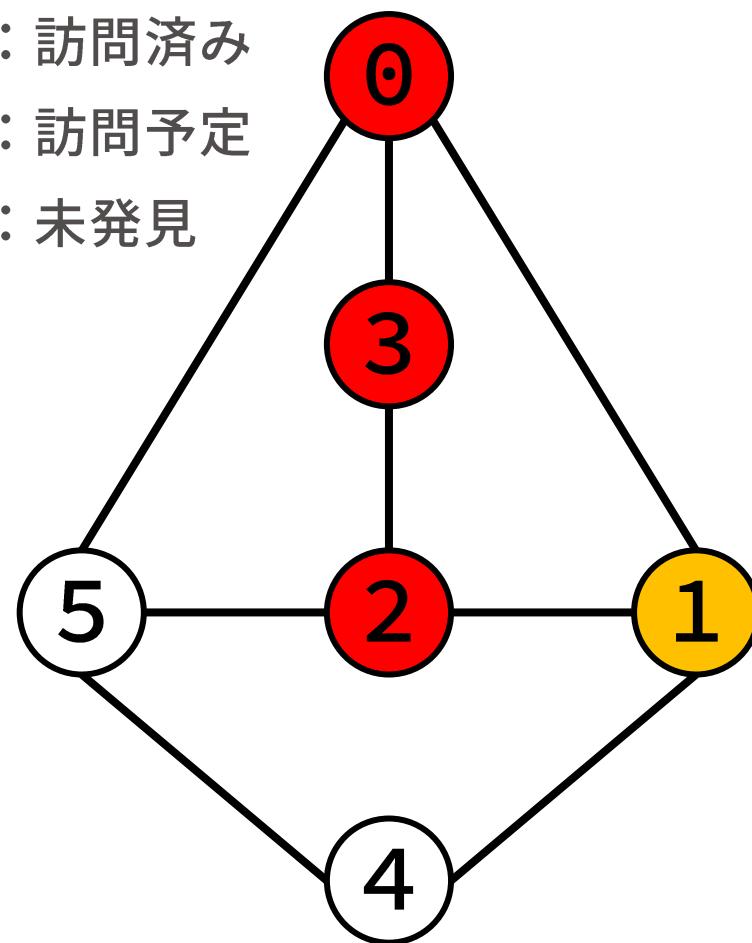
# バックトラック法 ハミルトンの閉路問題

2番を出発し、全ての頂点を1度だけ訪問

P159

枝切りについて

赤：訪問済み  
橙：訪問予定  
白：未発見



- (1) 2-3-0-5-4-1-2  
(2) 2-3-0-1-4-5-2

※(2)は探索済みとする

# バックトラック法 ハミルトンの閉路問題

2番を出発し、全ての頂点を1度だけ訪問

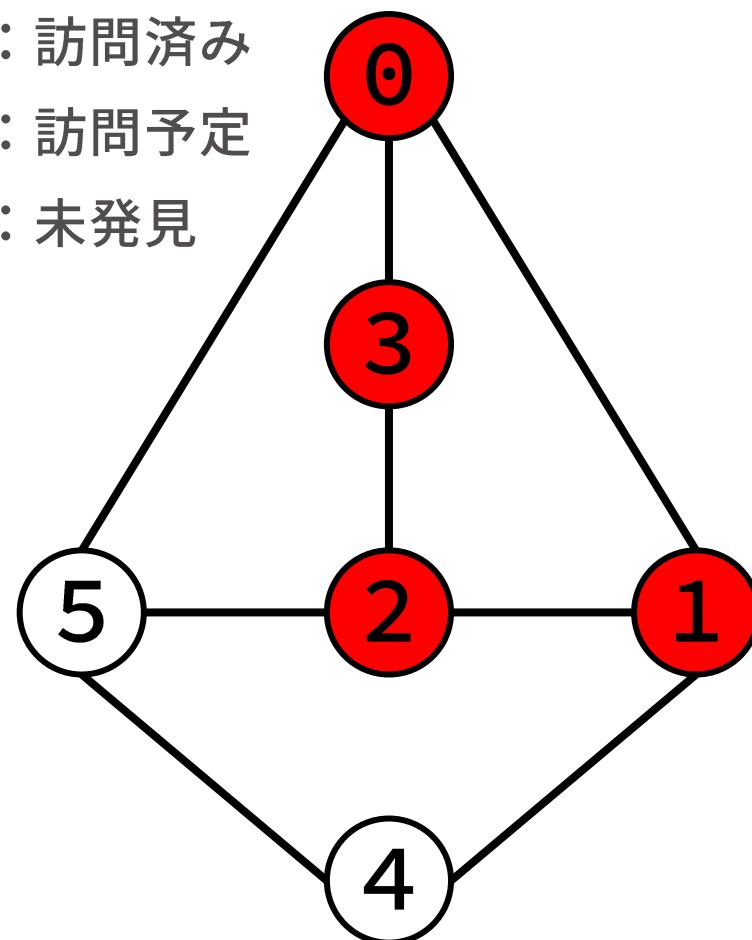
P159

枝切りについて

- (1) 2-3-0-5-4-1-2
- (2) 2-3-0-1-4-5-2

※(2)は探索済みとする

赤：訪問済み  
橙：訪問予定  
白：未発見



# バックトラック法 ハミルトンの閉路問題

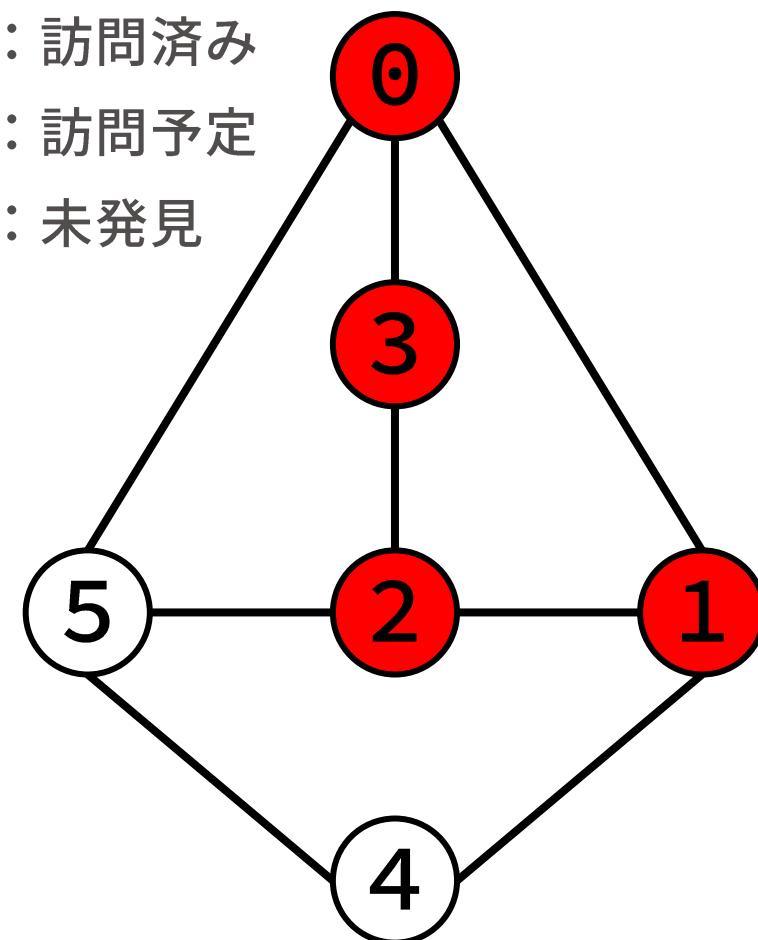
2番を出発し、全ての頂点を1度だけ訪問

P159

枝切りについて

- (1) 2-3-0-5-4-1-2
- (2) 2-3-0-1-4-5-2

※(2)は探索済みとする



- ②は制約上訪問不可能
- ④はすでに探索済み

# バックトラック法 ハミルトンの閉路問題

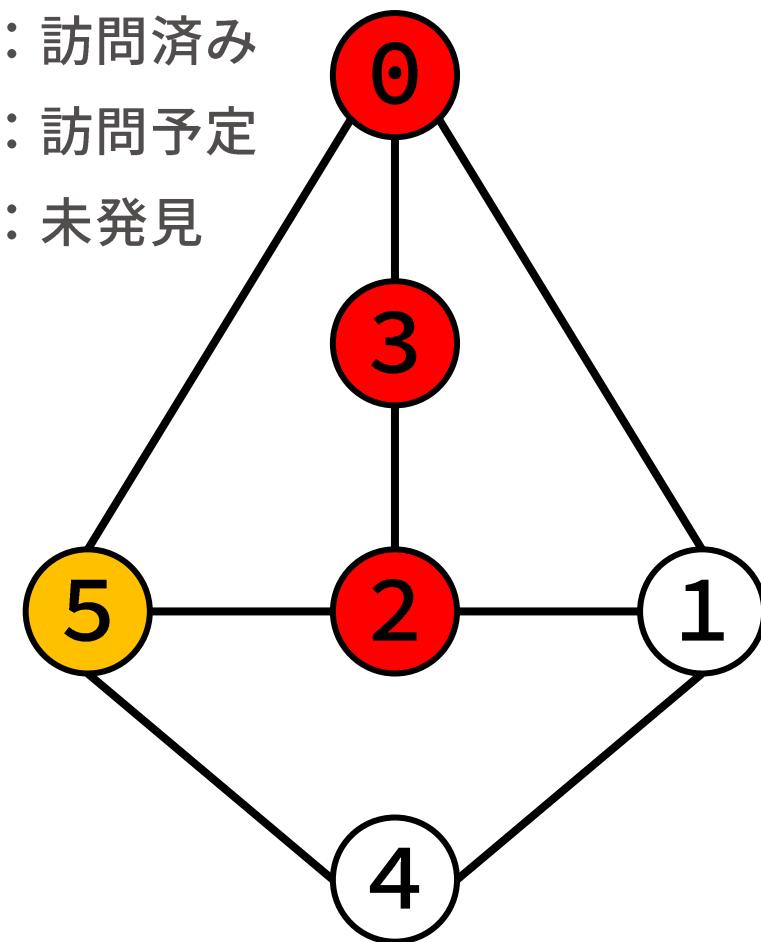
2番を出発し、全ての頂点を1度だけ訪問

P159

枝切りについて

- (1) 2-3-0-5-4-1-2
- (2) 2-3-0-1-4-5-2

※(2)は探索済みとする



②は制約上訪問不可能  
④はすでに探索済み  
→ 1歩戻り探索を行う

# バックトラック法 ハミルトンの閉路問題

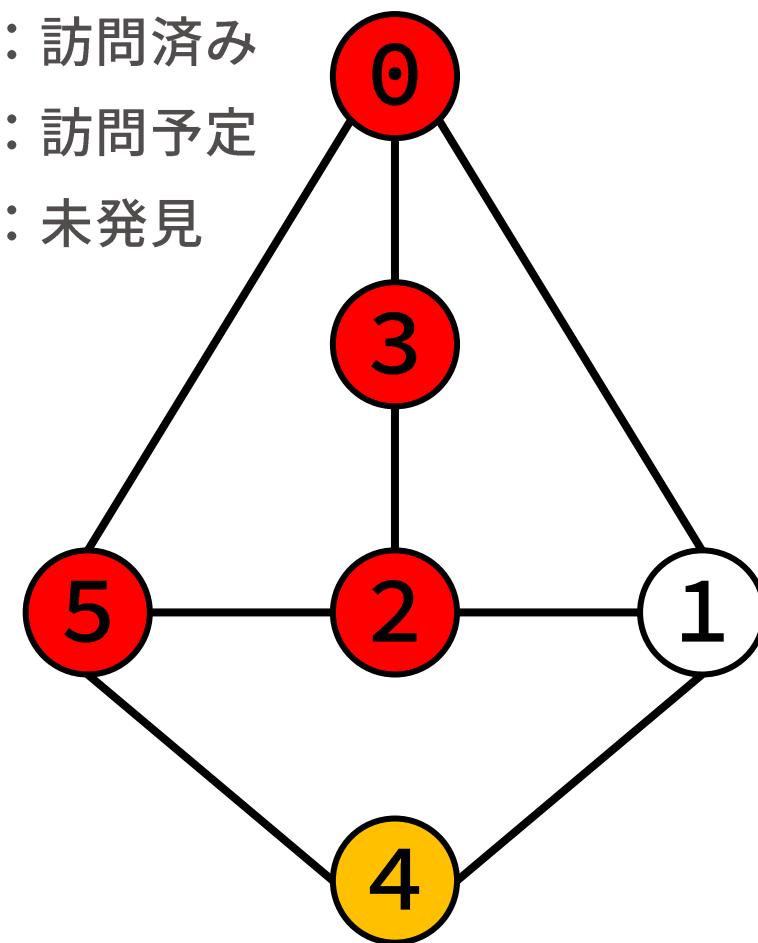
2番を出発し、全ての頂点を1度だけ訪問

P159

枝切りについて

- (1) 2-3-0-5-4-1-2
- (2) 2-3-0-1-4-5-2

※(2)は探索済みとする



途中で解が得られないと判明  
その先にある木は調べないように  
する  
→枝切り

# バックトラック法 ハミルトンの閉路問題

## アルゴリズム的なお話

P161~

```
#define N 6           // N:ノード数

// 無向グラフの行列表現
int a[N][N] = {
    {0, 1, 0, 1, 0, 1},
    {1, 0, 1, 0, 1, 0},
    {0, 1, 0, 1, 0, 1},
    {1, 0, 1, 0, 0, 1},
    {0, 1, 0, 0, 0, 1},
    {1, 0, 1, 1, 1, 0}};

int val[N] = {0, 0, 0, 0, 0, 0}; // ノードが探索済みかどうか識別するための配列
int start = 2;                // start:探索を開始するノード番号(0-5)ノード2から探索を開始
int id = 0;                   // id:探索済みのノード数を格納する変数
int route[N];                // 経路情報を保持するための配列
```

# バックトラック法 ハミルトンの閉路問題

## アルゴリズム的なお話

P161~

```
int main(void){  
    int i;  
  
    // ①  
    route[id] = start; // 経路情報をセットする id = 0 start = 2  
  
    // startノードから探索を行うため、探索のサブプログラムを呼び出す  
    visit(start);  
}
```

# バックトラック法 ハミルトンの閉路問題

## アルゴリズム的なお話

P161~

```
// バックトラック法による探索サブプログラム
void visit(int k){
    int i, t;

    // ②
    route[id] = k;           // route[]:探索経路をノード番号で示す変数
    val[k] = ++id;            // k:現在ノード位置
    // ノードkが探索済みであることを示す
```

# バックトラック法 ハミルトンの閉路問題

## アルゴリズム的なお話

P161~

```
// ③
// ノードを6個探索済みで,
// かつ, 現在ノード位置から探索開始位置にグラフが繋がっているならば
// この状態に至る経路が階の1つになる. 経路情報を出力する
if((id == 6) && (a[k][start] == 1)){ // 探索が最終まで辿り着いたかの確認
    printf("route=");
    for (i = 0; i < N; i++){
        printf("%3d", route[i]); // 経路情報を出力
    }
    printf("\n");
}
```

# バックトラック法 ハミルトンの閉路問題

## アルゴリズム的なお話

P161~

```
// ④
// 探索が途中状態にあるならば、現在ノードから次にたどり着けるノードを全て見つけ
// 条件に合致するノード探索を行うため、再帰的に探索プログラムを呼び出す
for (t = 0; t < N; t++){
    if(a[k][t] > 0){
        if(val[t] == 0){
            visit(t);
        }
    }
}
```

	[0]	[1]	[2]	[3]	[4]	[5]
route	2					
var	0	0	1	0	0	0
a[2][]	0	1	0	1	0	1

# バックトラック法 ハミルトンの閉路問題

## アルゴリズム的なお話

P161~

```
// ⑤
    // 探索条件を満足するノードがない場合は
    // 1つ前のノードにバックし、このノードから先のノードの探索を中止する
    route[id] = 0;
    id--;
    val[k] = 0;
```

# バックトラック法 ハミルトンの閉路問題

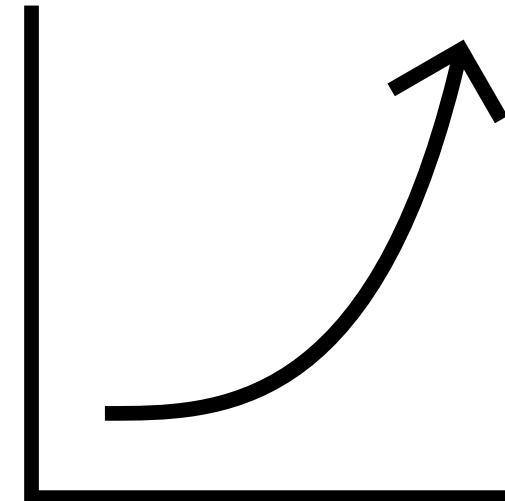
計算時間的なお話

P164

探索木の各項の平均： $t$ 個

探索の長さ： $N$

→頂点の数： $t^N$ 個



# バックトラック法 ハミルトンの閉路問題

計算時間的なお話

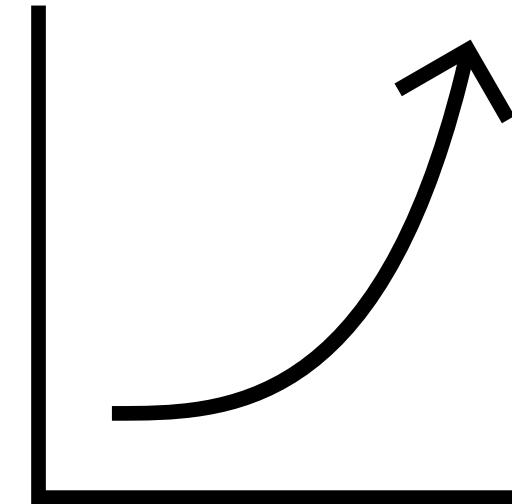
P164

探索木の各項の平均： $t$ 個

探索の長さ：  $N$

→頂点の数：  $t^N$  個

計算時間は**指数関数的**に増大



# バックトラック法 ハミルトンの閉路問題

計算時間的なお話

P164

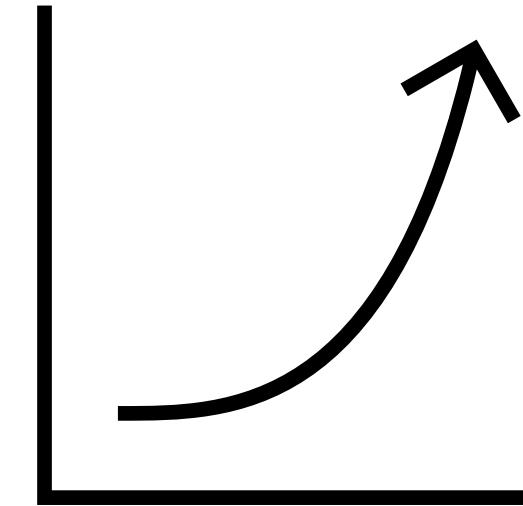
探索木の各項の平均： $t$ 個

探索の長さ：  $N$

→頂点の数： $t^N$  個

計算時間は**指数関数的**に増大

→ $t$ を小さくすれば計算時間減少



# お品書き

---

- 幅優先探索 p.151~159
- バックトラック法 p.159~164
- 文字列探索 p.170~177
  - 単純な文字列探索
  - BM法

# 文字列探索

P170

## テキスト文字列

0	1	2	3	4	5	6	7	8	9	10	11
M	O	J	I	R	E	T	U	T	A	N	S

## 照合文字列

E	T	U
---	---	---

Q：テキスト文字列に照合文字列はあるか？

# 文字列探索

P170

## テキスト文字列

0	1	2	3	4	5	6	7	8	9	10	11
M	O	J	I	R	E	T	U	T	A	N	S

## 照合文字列

E	T	U
---	---	---

Q：テキスト文字列に照合文字列はあるか？

A：配列の5番目

→ 文字列探索

# 文字列探索 単純な文字列探索

---

P170

テキスト文字列

M	O	J	I	R	E	T	U	T	A	N	S
---	---	---	---	---	---	---	---	---	---	---	---

照合文字列

E	T	U
---	---	---

# 文字列探索 単純な文字列探索

P170

テキスト文字列

M	O	J	I	R	E	T	U	T	A	N	S
---	---	---	---	---	---	---	---	---	---	---	---

ステップ1：最初の3文字比較 → 不一致

E	T	U
---	---	---

# 文字列探索 単純な文字列探索

P170

テキスト文字列

M	O	J	I	R	E	T	U	T	A	N	S
---	---	---	---	---	---	---	---	---	---	---	---

ステップ2：1文字ずらして比較 → 不一致

	E	T	U
--	---	---	---

# 文字列探索 単純な文字列探索

P170

テキスト文字列

M	O	J	I	R	E	T	U	T	A	N	S
---	---	---	---	---	---	---	---	---	---	---	---

ステップ3：1文字ずらして比較 → 不一致

		E	T	U
--	--	---	---	---

# 文字列探索 単純な文字列探索

P170

テキスト文字列

M	O	J	I	R	E	T	U	T	A	N	S
---	---	---	---	---	---	---	---	---	---	---	---

ステップ4：1文字ずらして比較 → 不一致

			E	T	U
--	--	--	---	---	---

# 文字列探索 単純な文字列探索

P170

テキスト文字列

M	O	J	I	R	E	T	U	T	A	N	S
---	---	---	---	---	---	---	---	---	---	---	---

ステップ5：1文字ずらして比較 → 不一致

				E	T	U
--	--	--	--	---	---	---

# 文字列探索 単純な文字列探索

P170

テキスト文字列

M	O	J	I	R	E	T	U	T	A	N	S
---	---	---	---	---	---	---	---	---	---	---	---

ステップ6：1文字ずらして比較 → **一致**

					E	T	U
--	--	--	--	--	---	---	---

# 文字列探索 単純な文字列探索

P170

# テキスト文字列

# M O J I R E T U T A N S

ステップ7：1文字ずらして比較 → 不一致

E T U

# 文字列探索 単純な文字列探索

P170

# テキスト文字列

# M O J I R E T U T A N S

## ステップ8：1文字ずらして比較 → 不一致

E T U

# 文字列探索 単純な文字列探索

P170

# テキスト文字列

# M O J I R E T U T A N S

ステップ9：1文字ずらして比較 → 不一致

E T U

# 文字列探索 単純な文字列探索

P170

# テキスト文字列

# M O J I R E T U T A N S

ステップ10：1文字ずらして比較 → 不一致

E T U

# 文字列探索 単純な文字列探索

P170

# テキスト文字列

# M O J I R E T U T A N S

ステップ11：テキスト文字列の最後に到達 → 探索終了

E T U

# 文字列探索 単純な文字列探索

---

アルゴリズム的なお話

P171

<手順1>

照合文字列とその文字数nを指定

```
// テキスト文字列の設定  
char text[] = "AIUEO",  
    // 照合文字列の設定  
    pattern[] = "GK";
```

# 文字列探索 単純な文字列探索

---

アルゴリズム的なお話

P171

<手順2>

テキスト文字列・照合文字列の照合位置を  $i, j$  とし,  
これらの値を 0 にする

```
int i, j;  
// i: テキスト文字列中の探索位置, 初期値0  
// j: 照合文字列中の探索位置, 初期値0  
i = j = 0;
```

# 文字列探索 単純な文字列探索

---

アルゴリズム的なお話

P171

<手順3>

テキスト文字列の最後か照合文字列の最後に達するまで  
以下の処理をする

<手順3-1>

テキスト文字列の*i*番目と照合文字列の*j*番目の文字を比較し  
一致するならば*i*と*j*にそれぞれ1を加える

不一致ならば,  $i = i - j + 1$ ,  $j = 0$

# 文字列探索 単純な文字列探索

---

アルゴリズム的なお話

P171

<手順3>

テキスト文字列の最後か照合文字列の最後に達するまで  
以下の処理をする

```
while (text[i] != '\0' && pattern[j] != '\0') {
```

¥0：終端文字

# 文字列探索 単純な文字列探索

アルゴリズム的なお話

P171

<手順3-1>

テキスト文字列の*i*番目と照合文字列の*j*番目の文字を比較し  
一致：*i*と*j*に1を加える 不一致：*i*=*i*-*j*+1, *j*=0

```
if (text[i] == pattern[j]){
    i++;
    j++;
}else{
    i = i - j + 1;
    j = 0;
}
```

# 文字列探索 単純な文字列探索

---

アルゴリズム的なお話

P171

<手順4>

比較が照合文字列の最後→テキスト文字列中に存在

照合文字列の先頭文字がテキスト文字列のどこにあるか  
位置を返す

```
if(pattern[j] == '¥0'){
    return i - j;
}
```

# 文字列探索 単純な文字列探索

---

アルゴリズム的なお話

P171

<手順5>

見つからなかった場合、-1を返す

```
return -1;
```

# お品書き

---

- 幅優先探索 p.151~159
- バックトラック法 p.159~164
- 文字列探索 p.170~177
  - 単純な文字列探索
  - BM法

# 文字列探索 BM法

簡略ボイヤー・ムーア法(BM法)

P173

ボイヤーさんとムーアさんが考案した  
文字列探索アルゴリズム



Robert S. Boyer



J Strother Moore

単純な文字列探索と違い**末尾**から比較

文字数がある程度の長さの場合**最も早い**文字列検索  
アルゴリズム

# 文字列探索 BM法

---

アルゴリズム的なお話

P173

テキスト文字列： MOJIRETUTANS

照合文字列： ETU

# 文字列探索 BM法

---

アルゴリズム的なお話

P173

テキスト文字列： MOJIRETUTANS

照合文字列： ETU

照合文字列の右端からの位置を表にする

※右端の文字は除外

文字	E	T	X
位置	2	1	

# 文字列探索 BM法

---

アルゴリズム的なお話

P173

照合文字列の右端からの位置を表にする

※右端の文字は除外

例えば、照合文字列が「ETETU」の時

右端省いて「ETET」

# 文字列探索 BM法

---

アルゴリズム的なお話

P173

照合文字列の右端からの位置を表にする

※右端の文字は除外

例えば、照合文字列が「ETETU」の時

右端省いて「ETET」

T → 1,3番目

E → 2,4番目

# 文字列探索 BM法

P173

アルゴリズム的なお話

照合文字列の右端からの位置を表にする

※右端の文字は除外

例えば、照合文字列が「ETETU」の時

右端省いて「ETET」

T → 1,3番目

E → 2,4番目

小さい値を取る



文字	E	T
位置	2	1

# 文字列探索 BM法

P173

アルゴリズム的なお話

テキスト文字列： MOJIRETUTANS

照合文字列： ETU

照合文字列の右端からの位置を表にする

※右端の文字は除外

文字	E	T	X
位置	2	1	

# 文字列探索 BM法

# アルゴリズム的なお話し

P173

<1>

テキスト文字列と照合文字列を左端を基準に揃える  
比較する 「J」と「U」 → 一致しない

0 1 2 3 4 5 6 7 8 9 10 11

M 0 J I R E T U T A N S

E T U

# 文字列探索 BM法

アルゴリズム的なお話

P173

<2>

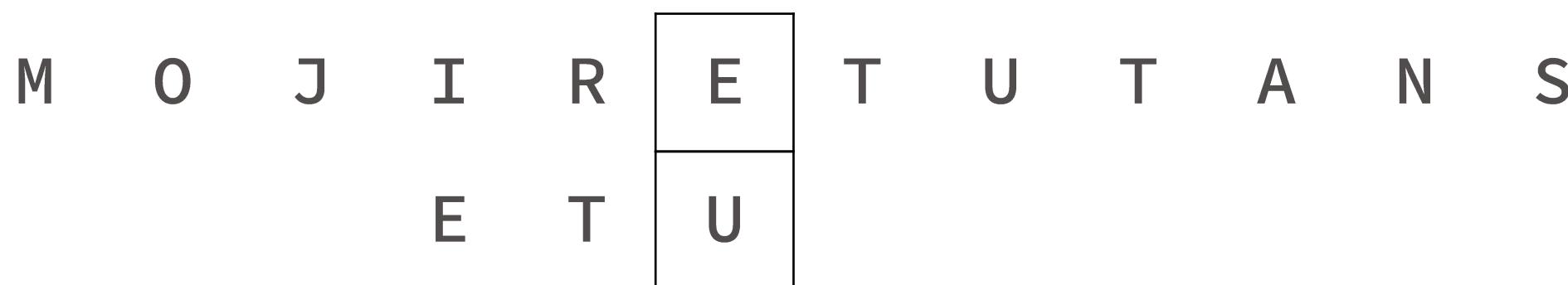
一致しないので照合文字列をずらす

ずらす文字数を表から引く

文字	E	T
位置	2	1

表中に「J」がないので、3文字分右にずらす

0 1 2 3 4 5 6 7 8 9 10 11



# 文字列探索 BM法

アルゴリズム的なお話

P173

<3>

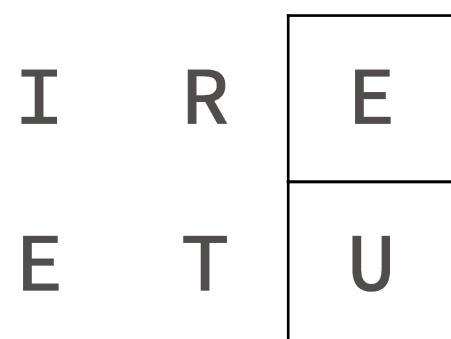
再度文字を比較する

「U」と「E」 → 一致しない

文字	E	T
位置	2	1

0 1 2 3 4 5 6 7 8 9 10 11

M O J I R E T U T A N S



# 文字列探索 BM法

アルゴリズム的なお話

P173

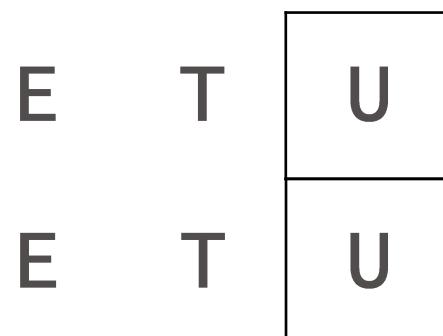
<4>

「E」は表中にあるので2文字分右に移動

文字	E	T
位置	2	1

0 1 2 3 4 5 6 7 8 9 10 11

M O J I R E T U T A N S



# 文字列探索 BM法

アルゴリズム的なお話

P173

<5>

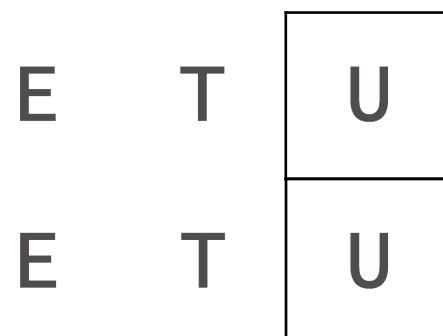
「U」と「U」を比較

→ 一致

文字	E	T
位置	2	1

0 1 2 3 4 5 6 7 8 9 10 11

M O J I R E T U T A N S



# 文字列探索 BM法

アルゴリズム的なお話

<6>

右端から順に左に向かって比較

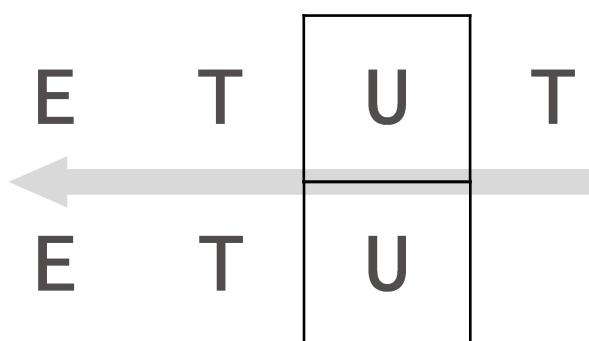
途中で一致しなければ右端に戻り，表中の数値分右に移動  
し探索を繰り返す 全て一致すれば探索を終了

P173

文字	E	T
位置	2	1

0 1 2 3 4 5 6 7 8 9 10 11

M O J I R E T U T A N S



# 文字列探索 BM法

プログラム的なお話

P174

<手順①>

テキスト文字列と照合文字列の定義

```
void main(){
    int p;
    char text[] = "MOJIRETUTANS¥0";
    char pattern[] = "ETU¥0";
    p = pos(text, pattern);
    if (p >= 0){
        printf("位置=%d¥n", p);
    }else{
        printf("見つかりません. ¥n¥n");
    }
}
```

# 文字列探索 BM法

プログラム的なお話

P174

<手順1>

変数や配列の定義

```
int i, j, k, len;
int skip[256]; // 256 = ASCIIコード
char c, tail;

// 文字列の長さ = 3
len = strlen((char*)pattern);

// 最後の文字を取得 = U
tail = pattern[len - 1];
```

# 文字列探索 BM法

---

プログラム的なお話

P174

// 次からvscode参照

# 文字列探索 BM法

---

計算量のお話

P177

BM法は文字数がある程度以上の長さの場合

**最も早い文字列検索アルゴリズム**

# 文字列探索 BM法

---

計算量のお話

P177

BM法は文字数がある程度以上の長さの場合

**最も早い文字列検索アルゴリズム**

→テキスト文字列中の文字の大部分を調べなくとも済むため

# 文字列探索 BM法

---

計算量のお話

P177

テキスト文字列長：  $n$

照合文字列長：  $m$

# 文字列探索 BM法

計算量のお話

P177

テキスト文字列長：  $n$

照合文字列長：  $m$

	最悪計算量
他のアルゴリズム	$O(n)$
BM法	$O(n)$

# 文字列探索 BM法

計算量のお話

P177

テキスト文字列長： $n$

照合文字列長：  $m$

	最悪計算量
他のアルゴリズム	$O(n)$
BM法	$O(n)$



$n/m$ 個の文字と比較すれば済む(ことがある)

# 文字列探索 BM法

---

計算量のお話

P177

テキスト文字列長： $n$

照合文字列長：  $m$

$m$ の長さが長いときに，計算時間が大幅に短縮できる

**可能性**がある

---

**EOF**